Diseño de Operadores de Mutación para Características de Sensibilidad al Contexto en Aplicaciones Móviles

Isyed de la Caridad Rodríguez Trujillo[1] y Macario Polo Usaola[2]

[1] Universidad de Concepción. Concepción, Chile
[2] Universidad de Castilla-La Mancha. Ciudad Real, España
isyedrodriguez@udec.cl
Macario.polo@uclm.es

Resumen. Este artículo presenta el diseño arquitectónico de un conjunto de operadores de mutación. Este diseño mejora el tiempo y coste de implementación de nuevos operadores respecto de la experiencia previa de los autores en el desarrollo de otras herramientas de mutación. El diseño, además, se está utilizando para la creación de operadores específicamente diseñados para reproducir artificialmente errores sobre las características de sensibilidad al contexto de aplicaciones móviles.

Palabras clave: Operadores de mutación, Tecnología móvil, Testing.

1 Introducción

El *mutation score* es el criterio de cobertura que se utiliza para guiar la construcción de casos de prueba, en pruebas mediante mutación. Un test suite alcanza un 100% de *mutation score* cuando encuentra todos los errores artificiales que se han insertado en los mutantes, que son copias del sistema que se está probando (el SUT, *System Under Test*). Los errores artificiales, se insertan mediante *operadores de mutación*, que son elementos especializados en la introducción de un tipo de cambio (reemplazo, inserción o eliminación). En este sentido, la calidad del test suite y del SUT va a depender de la naturaleza de estos errores. Si un test suite encuentra todos los errores artificiales, entonces es un test suite de alta calidad; además, si este mismo test suite no encuentra ningún error en el SUT, entonces se puede decir que el SUT está libre de errores [1] o, al menos, que está libre de los errores artificiales insertados.

La mutación confía en dos principios importantes [2]:

- La *hipótesis del programador competente*, según la cual un programa escrito por un "programador competente" será casi perfecto, de modo que diferirá de la versión correcta por fallas relativamente simples.
- El *efecto acoplamiento*, que afirma que el descubrimiento de errores simples permite el hallazgo de errores más complejos. La validez de este efecto se ha comprobado parcialmente con mutantes de orden superior, siendo Harman y colabores quienes más han trabajado en ello [3].

En este artículo se menciona un conjunto de operadores de mutación específicamente diseñados para probar las características de sensibilidad al contexto de aplicaciones móviles, como Android. Una línea de trabajo, en la que se ha puesto especial cuidado, es el diseño arquitectónico de los operadores de mutación, para poder crear con facilidad nuevos operadores. Por ello, un objetivo importante del diseño de los operadores, es su implementación, de manera que su arquitectura permita la reutilización y el enriquecimiento sencillo del juego de operadores. En la sección 2 se presenta el diseño arquitectónico de los operadores, que es la contribución principal de este trabajo y en la sección 3 se presentan algunas conclusiones.

2 Diseño Arquitectónico de los Operadores

Todos los operadores definidos son especializaciones de una clase abstracta *Operator* (en la Figura 1 se muestra la estructura de esta clase para métodos, que es el tipo de miembro en el que se está trabajando en estos momentos), en cuyos objetos se guarda el nombre del fichero *.class* que se va a procesar y el grupo al que pertenece el operador (campo *family*). La clase ofrece la operación concreta *methodIsMutable*, que devuelve true si el método posee alguna instrucción mutable. Así, *methodIsMutable* recorre la lista de instrucciones *bytecode* correspondiente, preguntando en cada instrucción si ésta es mutable (operación abstracta *instructionIsMutable*, de la librería ASM).

El método *generateMutants* guarda en ficheros cada uno de los mutantes producidos por el operador, mediante llamadas a *performMutation* que, a su vez, llama a las operaciones abstractas *mutate*.

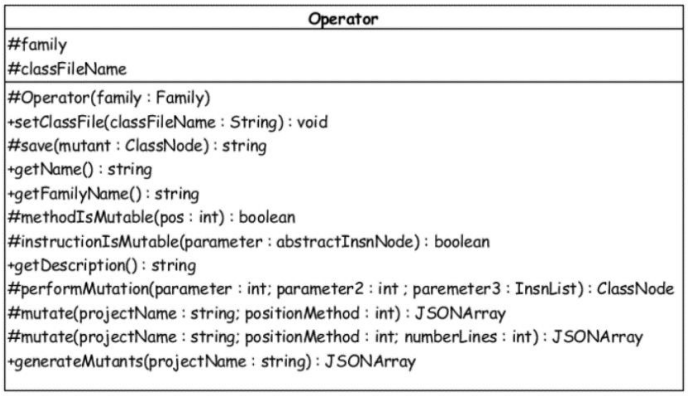

Fig. 1. Miembros de la clase abstracta *Operator*.

La inserción de un error artificial a nivel de *bytecode* puede suponer el reemplazo, la eliminación o la inserción de una instrucción. Esta clasificación de operadores tiene su representación en el sencillo diseño de la Figura 2.

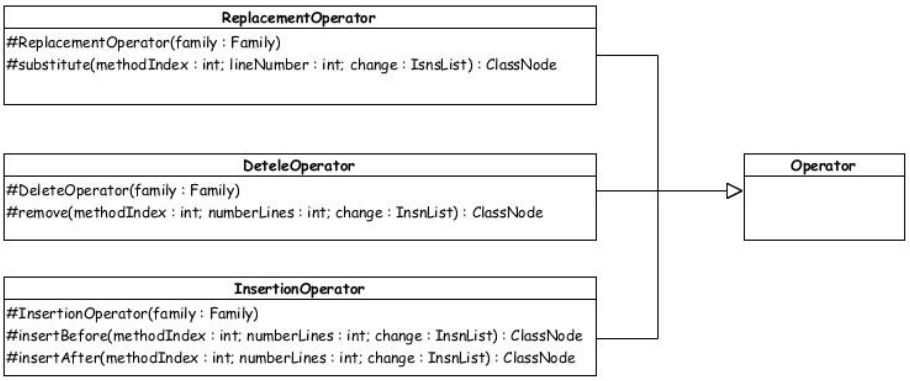

Fig. 2. Tres especializaciones de *Operator*.

2.1 Operadores de Sensibilidad al Contexto

En el estado del arte, existen operadores de mutación específicos para aplicaciones Android [4],[5]. Sin embargo, todavía no han sido considerados todos los aspectos de estas aplicaciones. Por lo que, es necesario el desarrollo de nuevos operadores que permitan reproducir errores representativos de esta tecnología. Algunos de los operadores, implementados en la herramienta BacterioWeb (que estamos desarrollando) y que reproducen errores de sensibilidad al contexto son: WSR (*Wrong Sensor Requested*), WSS (*Wrong Speed In Sensor*), SSN (*Sensor Sends Null*), ULS (*Unregister Listener Sensor*), CI (*Coordinates Interchange*), FEC (*Forget Editor Commit*), SNRR (*Servidor Not Return Response*), DIOMTO (*Database Is Opened More Than Once*) y DSNI (*Driver Sensor Not Initialized*).

A continuación se explica brevemente tres operadores, conforme las especializaciones de la clase *Operator* a la que pertenecen:

Operador de Reemplazo. WSR (*Wrong Sensor Requested o Sensor incorrecto solicitado*). Este error aparece cuando el programador solicita un sensor diferente al necesario. Por ejemplo, en lugar del *TYPE_ACCELEROMETER*, solicita el *TYPE_LINEAR_ACCELERATION*. Como el cambio para reproducir este error solo consiste en sustituir el valor de la constante que representa el sensor, el operador de mutación correspondiente para este error es una especialización de *ReplacementOperator* que se debe aplicar cuando se llama al método *getDefaultSensor*.

Operador de inserción. CI (*Coordinates Interchange o Coordenadas Intercambiadas*). Uno de los errores informados con respecto al uso de la pantalla proviene de eventos inesperados del usuario, como tocar dos veces en un widget o girar el dispositivo. Este evento pro-duce un cambio en las coordenadas donde toca el usuario: el centro de la pantalla, en lugar de estar en (x, y), pasa a estar en (y, x). Dependiendo de la operación donde se recolecta el evento, los datos en el evento pueden llegar a un tipo diferente de objeto. Por ejemplo, el método *onTouchEvent* de la clase *View*, recibe un objeto *MotionEvent*. Para simular el cambio de coordenadas, basta con agregar una llamada al método *setLocation (float, float)* como la primera línea de cualquier implementación de *onTouchEvent*, intercambiando las coordenadas con *getX* y *getY*. La inserción de esta

instrucción específica de Java requiere la inserción de seis nuevas instrucciones de *bytecode*.

Operador de eliminación. ULS (*UnregisterListenerSensor o Eliminar el sensor registrado*). El operador ULS simula cuando el programador olvida liberar un sensor, provocando en algunos casos baja batería. Reproducir este error consiste en eliminar la llamada a *unregisterListener (SensorListener).* A nivel de bytecode, esto significa eliminar todas las instrucciones entre *LINENUMBER* (instancia de *LineNumberNode*).

3 Conclusiones

Se han presentado los primeros frutos de un trabajo de investigación que se lleva a cabo entre las universidades de Castilla-La Mancha y de Concepción. Mientras que el objetivo final es el desarrollo de una herramienta web para automatizar las pruebas de mutación, en este trabajo nos hemos centrado en la descripción del diseño arquitectónico de operadores de mutación para sensibilidad al contexto en aplicaciones móviles. Comparado con el esfuerzo que se ha requerido para la implementación de operadores en otras herramientas, este nuevo diseño reduce mucho el coste de implementación de nuevos operadores.

Agradecimientos

Este trabajo está parcialmente financiado por el proyecto TESTIMO - “Mejora del proceso de testing de software en base a sus tareas manuales", Junta de Comunidades de Castilla-La Mancha. Conjuntamente, con el apoyo de la Beca de Doctorado Nacional CONICYT del Gobierno de Chile, otorgada a Isyed de la Caridad.

Referencias

1. Usaola MP, Mateo PR (2010) Mutation Testing Cost Reduction Techniques: IEEE Int Conf Softw Test - ICST 27:80–86 . doi: 10.1109/MS.2010.79
2. Offutt AJ (1992) Investigation of the software testing coupling effect. ACM Trans Softw Eng Methodol 1:3–18
3. Harman M, Jia Y, Reales Mateo P, Polo M (2014) Angels and Monsters: An Empirical Investigation of Potential Test Effectiveness and Efficiency Improvement from Strongly Subsuming Higher Order Mutation. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE ’14. pp 397–408
4. Deng L, Offutt J, Ammann P, Mirzaei N (2017) Mutation operators for testing Android apps. Inf Softw Technol 81:154–168 . doi: 10.1016/j.infsof.2016.04.012
5. Deng L, Offutt J, Samudio D (2017) Is Mutation Analysis Effective at Testing Android Apps? 2017 IEEE Int Conf Softw Qual Reliab Secur 86–93 . doi: 10.1109/QRS.2017.19